Jurnal Ilmiah Widya Teknik
Volume 15 Nomor 2 2016
ISSN 1412-7350

Kinerja Kode Rapid Tornado Dengan Reed-Solomon *Precode*

Valentino Pradnya Mahinda, Jonathan Aristo, Lydia Sari*
Prodi Teknik Elektro, Fakultas Teknik, Universitas Katolik Indonesia Atma Jaya
Jl. Jend. Sudirman Kav. 51, Jakarta 12930
*Penulis korespondensi, e-mail: lydia.sari@atmajaya.ac.id

ABSTRCT

Rapid Tornado code is constructed from a Luby Transform code concatenated with a precode. This code is used to protect data sent over an erasure channel. An erasure channel will typically corrupt a number of transmitted data, causing the loss of several information symbols. Despite this loss, the transmitted data can still be reconstructed by the receiver if Rapid Tornado or Luby Transform code is used. In this paper, an encoding and decoding process using Rapid Tornado and Reed-Solomon precode is simulated with 9 information symbols sent over an erasure channel. The simulation result is compared to that of a Luby Transform code. It is shown that Rapid Tornado enables the reconstruction of data despite the loss of codewords transmitted over an erasure channel.

Kata kunci : *Binary Erasure Channel, Rapid Tornado, Luby Transform, degree distribution*

I. Pendahuluan

Meluasnya penggunaan internet sejak dekade 1990an telah menunjukkan bahwa kanal *Binary Erasure Channel* (BEC) yang sebelumnya dipandang sebagai model teoretis, adalah kanal yang dapat ditemui dalam kondisi riil. Di jaringan internet, data dikirimkan dalam bentuk paket-paket yang masing-masing dilengkapi dengan alamat pengirim dan penerima, serta penanda yang menunjukkan posisi relatif sebuah paket dalam aliran data. Setiap paket akan menempuh rute yang mungkin berbeda-beda dari pengirim ke penerima. Kehilangan paket dapat terjadi karena berbagai hal, misalnya penumpukan data di *buffer* pada *router-router* yang digunakan [1]. Kemungkinan terhapusnya paket data dalam jaringan internet menunjukkan bahwa jaringan Internet merupakan contoh kanal BEC riil.

Berbagai penelitian telah dilakukan untuk meningkatkan kehandalan data pada Internet. Pada umumnya digunakan protokol komunikasi tertentu untuk menjamin kehanddalan data, misalnya dengan menggunakan TCP/IP agar paket yang tidak memperoleh *receiver acknowledgment* setelah dikirimkan dapat ditransmisikan ulang. Namun demikian telah diketahui bahwa protokol-protokol komunikasi tersebut tidak memiliki kinerja optimal pada kasus-kasus tertentu misalnya saat transmisi data dari satu pengirim ke penerima jamak [1], [2]. Metode lain untuk meningkatkan kehandalan data adalah dengan penggunaan *forward error correction*, misalnya menggunakan skema pengkodean Reed-Solomon, suatu kode blok yang umum digunakan untuk mengatasi permasalahan kanal *erasure* [3]. Sebuah kode Reed-Solomon (N,K) dengan ukuran alfabet $q = 2^L$ memiliki sifat ideal dimana bila sembarang K dari N simbol yang ditransmisikan dapat tiba di penerima, maka K simbol *original* akan dapat direkonstruksikan. Namun demikian kode Reed-Solomon yang berdiri sendiri memiliki kelemahan yaitu hanya praktis diterapkan untuk nilai K, N dan q yang kecil [3].

Kode Luby Transform (LT) diajukan untuk mengatasi permasalahan hilangnya data pada kanal *erasure* [4], [5]. Kode ini termasuk dalam keluarga kode Fountain, dimana *codeword* dibentuk dengan cara menggabungkan beberapa simbol masukan. Semakin banyak *codeword* yang dihasilkan dari kombinasi simbol-simbol masukan, maka semakin besar peluang penerima untuk merekonstruksikan simbol yang dikirim walaupun terjadi *erasure*. Enkoder LT dapat membangkitkan *codeword* sebanyak yang diperlukan untuk mendekodekan sejumlah simbol informasi. Pada skema pengkodean LT, pengirim tidak memerlukan *acknowledgement* dari pihak penerima. Sifat ini merupakan keunggulan skema LT terutama jika digunakan untuk sistem *multicast*, karena jumlah overhead dapat diminimalisir. Pembangkitan *codeword* pada kode LT dilakukan mengikuti *degree distribution* tertentu. Setiap kali sebuah *codeword* dibangkitkan dalam skema pengkodean LT, dilakukan *sampling* distribusi bobot yang menghasilkan sebuah bilangan bulat d yang bernilai antara 1 hingga k, dimana k adalah jumlah simbol masukan. Kemudian, d simbol masukan acak dipilih dan nilainya dijumlahkan untuk memperoleh *codeword*. Keberhasilan proses *decoding* tergantung hanya pada *degree distribution* dari *codeword* [1].

Kode Rapid Tornado (Raptor) adalah salah satu jenis kode Fountain yang merupakan penggabungan antara *forward error correction* (sebagai *pre-code*) dan kode LT. Skema pengkodean yang digunakan sebagai *pre-code* pada penelitian ini adalah Reed-Solomon. Keunggulan skema Raptor adalah laju kode yang adaptif sehingga dapat disesuaikan dengan kondisi kanal. Dalam pengembangan selanjutnya, sifat adaptif tersebut dapat dimanfaatkan untuk pengaturan konsumsi energi perangkat yang menggunakan kode Raptor.

Makalah ini membahas kinerja kode Raptor dengan *pre-code* Reed-Solomon pada kanal BEC yang dimodelkan dengan penghapusan acak sejumlah *codeword*. Bagian kedua makalah ini memaparkan pembentukan *codeword* Raptor, proses *decoding* kode Raptor, serta perhitungan *degree distribution*. Bagian ketiga memaparkan distribusi *degree* untuk kode LT. Bagian ketiga membahas hasil dari simulasi pengkodean menggunakan piranti lunak Matlab sedangkan kesimpulan diberikan pada bagian terakhir.

II. Landasan Teori

II.1 Konstruksi Kode

II.1.1 Kode Reed-Solomon

Proses pengkodean Reed-Solomon melibatkan Galois Field $GF(p^m)$.Tahap pertama dalam pengkodean adalah membuat bit paritas pada data yang dikirim. Pembuatan paritas bisa dilakukan dengan rumus [6],[7]:

$CK(x) = \left(X^{n-k}.M(x)\right) mod\ g(x)$ … (1)

dengan :

CK(x) = *Parity check*
n = bit total dalam *codeword*
k = bit total dalam pesan
t = jumlah bit yang dapat di koreksi
M(x) = data yang dikirim
g(x) = generator RS

Generator RS memiliki bentuk sebagai berikut [6],[7]:

$g(x) = (x + \alpha).(x + \alpha^2) \ldots (x + a^{n-k})$ … (2)

Codeword dari Reed Solomon bisa didapatkan dengan perhitungan [5]:

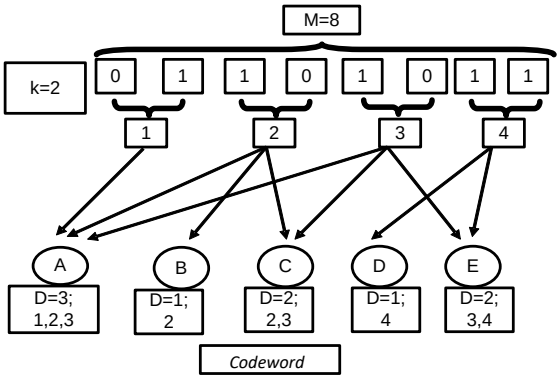
$C(x) = \left(X^{n-k}.M(x)\right) + CK(x)$ … (3)

II.1.2 Kode LT

Secara ringkas proses pembentukan *codeword* LT adalah sebagai berikut [3]:

1. Simbol masukan dibagi atas *k* bit
2. Tentukan *degree d* untuk setiap *codeword*.
3. Pilih *d* simbol masukan secara acak untuk kemudian digabungkan dengan operasi XOR membentuk sebuah *codeword.*
4. *Codeword* ditransmisikan ke penerima, setiap *codeword* mengandung *k* atau kelipatan *k* bit informasi. Adanya bit-bit yang redundan dalam *codeword* yang berbeda memungkinkan penerima merekonstruksi simbol masukan bahkan bila terjadi hilangnya *codeword* dalam kanal *erasure*.

Gambar 1 menunjukkan ilustrasi pengkodean LT. Data yang berjumlah 8 bit dibagi dengan *k* sehingga berjumlah 4 blok. Selanjutnya dilakukan operasi *XOR* antar blok-blok tersebut untuk menghasilkan *codeword.* Pada contoh yang diilustrasikan pada Gambar 1, blok pertama akan di-*XOR* dengan blok kedua dan ketiga sehingga menghasilkan *codeword* A. Blok kedua dan ketiga akan di-*XOR* sehingga menghasilkan *codeword* C. *Codeword* E dihasilkan dari proses *XOR* antara blok ketiga dan keempat.



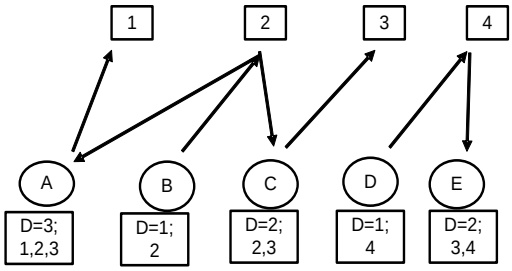
Gambar 1. Pengkodean LT

Dari contoh tampak bahwa *codeword* B dan *codeword* D hanya dihasilkan dari satu blok saja. Ini adalah syarat agar hasil pengkodean *Luby Transform* dapat didekodekan. Jumlah blok yang di-*XOR* untuk menghasilkan *codeword* disebut *degree distribution (d)*. *Codeword* LT dikirimkan melalui BEC yang bersifat dapat menghilangkan data. Untuk bisa mendekodekan kode LT, *decoder* harus mengetahui tetangga-tetangga dari setiap *codeword.* Informasi ini dapat dikirimkan dengan beberapa cara. Misalnya, pemancar dapat mengirimkan sebuah paket data yang terdiri atas *codeword* dan daftar tetangganya. Metode lainnya adalah *encoder* dan *decoder*

menggunakan generator angka acak dengan *seed* yang sama, dan decoder menentukan tetangga-tetangga dari setiap *codeword* dengan cara membangkitkan kombinasi linier acak yang di sinkronkan dengan enkoder.
Prosesnya *decoding* LT adalah sebagai berikut :

1) *Release*. Semua *codeword* yang mempunyai *degree* satu, yaitu *codeword* yang sama dengan simbol informasi, di-*release* untuk meng-*cover* tetangganya
2) *Cover*. *Codeword* yang telah di-*release* akan meng-*cover* simbol informasi tetangganya yang mempunyai *degree* dua. Hal ini berarti satu *codeword* yang telah berhasil di-*decode* akan memungkinkan proses *decode release* dari satu *codeword* lainnya. Pada langkah ini, simbol masukan yang sudah di-*cover* tapi belum diproses akan dikirim ke *ripple*. *Ripple* adalah kumpulan simbol informasi yang telah di–*cover* tapi belum diproses yang diperoleh dari iterasi-iterasi sebelumnya.
3) Proses. Satu simbol informasi dalam *ripple* dipilih untuk diproses. *Edge* yang menghubungkan simbol informasi tersebut ke *codeword* tetangganya akan dibuang dan nilai setiap *codeword* berubah tergantung simbol informasi. Simbol informasi yang telah di proses akan di keluarkan dari *ripple*

Gambar 2 mengilustrasikan proses *decoding* LT. Proses *decoding* LTc mempunyai syarat awal yaitu harus dimulai dari 1 simbol yang mempunyai *degree distribution* (*d)* yang bernilai 1 (*Release*).



Gambar 2. Proses *decoding* LT

Proses dimulai dari *codeword* B, apabila *codeword* B diketahui maka komponen 2 diketahui dan Komponen 4 bisa didapat dari *codeword* D (*Release*). Kemudian dilanjutkan dengan melakukan perhitungan logika XOR terhadap 2 dan *codeword* C dan akan menghasilkan komponen 3 (*Cover*). Kemudian setelah mengetahui komponen 2 dan 3, bisa didapat komponen 1 dengan melakukan perhitungan XOR antara 2, 3 dan A yang mempunyai *degree distrbution* hanya 1.

Proses *decoding* berlanjut dengan mengulang langkah-langkah tersebut di atas. Hanya *codeword* dengan *degree* satu yang dapat memicu terjadinya langkah-langkah tersebut di atas. Dengan demikian penting dipastikan bahwa selalu terdapat *codeword* dengan *degree* satu untuk di-*release* agar proses dekode dapat berjalan. Simbol informasi dalam *ripple* dapat mengurangi *degree* simbol *decoding*. Simbol-simbol informasi dalam *ripple* akan terus menghasilkan *codeword* dengan *degree* satu setelah tiap iterasi dan sebagai konsekuensinya proses *decoding* akan selesai setelah *ripple* kosong. Proses *decoding* berhasil bila semua simbol informasi dapat di-*cover*. Distribusi *degree* dari *codeword* dianalisis bedasarkan ukuran *ripple*.

II.2 Distribusi Degree pada Kode LT

Kode LT tidak memiliki laju tertentu. Karakter kode yang diinginkan adalah yang mempunyai probabilitas keberhasilan yang setinggi mungkin, namun mempunyai jumlah *codeword* yang sedikit. Dalam istilah proses LT, karakter tersebut dapat dinyatakan sebagai:

- Laju *release* dari *codeword* rendah agar *ripple* terjaga tetap berukuran lebih kecil dan mencegah adanya *codeword* yang tidak diperlukan.
- Laju *release* dari *codeword* cukup tinggi agar *ripple* tidak kosong.

Dengan demikian, distribusi *degree* dari *codeword* harus dirancang dengan seksama agar tercapai *tradeoff* yang seimbang. Ini adalah alasan pentingnya distribusi *degree e*pada kode LT.

Sebagai contoh, distribusi All At Once (P_1=1 dan P_d=0 untuk d=2,3,...) memerlukan *codeword* yang memiliki satu tetangga. Setiap *codeword* yang tiba di penerima dapat langsung me-*recover* simbol informasi yang terkait dengannya. Tetapi, bila ada satu *codeword* yang terhapus, maka simbol informasi yang terkait dengannya tidak dapat di-*recover*. Untuk mencegah terjadinya kegagalan *decoding*, pemancar harus mengirim *codeword* berjumlah lebih besar dari *k*. Distribusi ini menimbulkan banyak *codeword* yang sebenarnya tidak diperlukan. Secara teoretis, distribusi *degree* yang optimum untuk LT adalah distribusi Robust Soliton yang dikembangkan dari distribusi Ideal Soliton. Kedua distribusi tersebut akan dijelaskan pada subbab selanjutnya.

II.2.1 Distribusi Ideal Soliton

Distribusi ideal soliton menunjukkan kinerja yang ideal dalam jumlah *codeword* yang diperlukan untk meng-*cover* data. Distribusi ideal Soliton diberikan oleh [5].

$$\oplus_1 = \frac{1}{k} \ldots (4)$$

$\oplus_i = \frac{1}{i(i-1)}$ … (5)
dimana i = 1, …, k.
Untuk distribusi ideal soliton, $r(L) = \frac{1}{k}$, dengan L=1,...k.

Parameter K(L) bernilai sama untuk semua L, sehingga distribusi ideal Soliton menghasilkan probabilitas yang sama. Distribusi ideal Soliton memungkinkan diperlukannya hanya k *codeword* untuk meng-*cover* k simbol informasi dan tepat satu simbol *encoding* akan di-*release* setiap kali sebuah simbol informasi diproses. Ukuran *ripple* pada distribusi ini selalu tetap. Dengan demikian, tidak pernah ada *codeword* yang tidak diperlukan, dan *ripple* tidak akan pernah kosong.

Namun demikian dalam praktek, Distribusi Ideal Soliton menunjukkan kinerja yang buruk. Variasi yang kecil sekali akan dapat mengakibatkan kosongnya *ripple* di tengah proses *decoding*, sehingga terjadi kegagalan *decoding*. Untuk mengatasi permasalahan ini, digunakan Distribusi Robust Soliton.

II.2.2 Distribusi Robust Soliton

Parameter R dan ukuran *Ripple* dan S melambangkan probabilitas kegagalan yang diperbolehkan. Distribusi Robust Soliton M diberikan oleh dua distribusi $\oplus$ dan T, yaitu
$M_i = (\oplus_i + T_i)/\beta$ … (6)
Dimana $\oplus$ adalah Distribusi Ideal Soliton dan T dinyatakan oleh

$$T_i = \begin{cases} \frac{R}{ik} \; untuk \; i = 1, \dots, \frac{k}{R} - 1 \\ \frac{R \ln\left(\frac{R}{\delta}\right)}{k} \; untuk \; i = \frac{k}{R} \\ 0 \; untuk \; i = \frac{k}{R} + 1, \dots, k \end{cases} \quad \dots (7)$$

dan $\beta = \sum_i (\oplus_i + T_i)$ melambangkan faktor normalisasi.

Ide dasar dari Distribusi Robust Soliton adalah distribusi T yang meningkatkan ukuran *ripple* ditambahkan ke distribusi Ideal Soliton sehingga distribusi *degree* yang dihasilkan memiliki ukuran *ripple* yang lebih besar dari P namun probabilitas *Release* yang seragam tetap terjaga. Misalkan jumlah *codeword* adalah $n = k.\beta = k\sum_{i=1}^{k}(\oplus_i + T_i)$. Proses *decoding* dimulai dengan ukuran *ripple* $k(\oplus_i + T_i) = 1 + R$. Dalam proses *decoding*, setiap iterasi memproses satu simbol informasi, yang berarti *ripple* harus bertambah 1. Saat L simbol informasi belum diproses, simbol tersebut memerlukan L/(L-R) *codeword* yang telah di-*release* untuk menambah satu simbol ke *ripple*.

Laju *Release* dari *codeword* dengan *degree* i untuk $i=k/L$ membentuk bagian yang tetap dari laju *release* saat L simbol informasi belum diproses. Jadi, jumlah *codeword* dengan $degree i = \frac{k}{L}$ harus sebanding dengan
$\frac{1}{i(i-1)}\frac{L}{L-R} = \frac{k}{i(i-1)(k-iR)} = \frac{1}{i(i-1)} + \frac{R}{(i-1)(k-iR)}$ … (8)
Untuk $i=k/R$, T_i menjamin bahwa semua simbol informasi akan di-*cover* dan berada di dalam *ripple*, jadi $L=R$.
Dengan asumsi sebuah *Random Walk* sepanjang k akan menyimpang dari nilai rata-rata sejauh lebih dari $\ln\left(\frac{k}{\delta}\right)\sqrt{k}$ dengan probabilitas maksimal δ, ukuran *ripple* dapat dinyatakan sebagai
$R = c.\ln\left(\frac{k}{\delta}\right)\sqrt{k}$ … (9)
untuk konstanta c>0 sehingga probabilitas keberhasilan *decoding* lebih besar dari $1 - \delta$.

III. Hasil Simulasi

Simulasi kode Raptor dengan *pre-code* Reed-Solomon dilakukan dengan piranti lunak Matlab. Simulasi diawali dengan pembangkitan bit acak yang dilakukan secara manual berjumlah 9 bit informasi yang kemudian dikodekan menggunakan kode Reed-Solomon menjadi 15 bit informasi *codeword* Reed-Solomon. *Codeword* tersebut akan diproses oleh enkoder Luby Transform untuk menjadi *codeword* LT sepanjang 30 bit. Simulasi ini dilakukan sebanyak 50 kali. Hasil simulasi ditampilkan pada Tabel 1.

Sebagai pembanding, dilakukan simulasi untuk kode LT, dengan dengan simbol masukan yang berjumlah sama dengan simulasi kode *Rapid Tornado* yaitu 9 bit. Berbeda dengan simulasi kode Raptor, simulasi kode LT menggunakan matriks generator berukuran 9 baris 18 kolom dengan perbandingan yang sama dengan matriks generator pada kode Raptor yaitu 1 berbanding 2. Simulasi ini dilakukan sebanyak 50 kali. Hasil simulasi kinerja kode LT diberikan pada Tabel 2.

Tabel 1. Kinerja Kode Raptor dengan Reed-Solomon *Pre-code* Berdasarkan Simulasi

# Percobaan	Bit Informasi	Bit Hasil Dekode *Rapid Tornado*	Jumlah kesalahan dekode *Rapid Tornado* (bit)
1	000 000 001	000 000 000	1
2	000 000 011	000 000 000	2
3	000 000 111	001 000 101	2
4	000 000 100	000 000 000	1
5	000 000 101	001 001 100	3
6	000 000 010	010 000 010	1
7	000 000 110	010 100 110	2
8	000 001 000	100 001 000	1
9	000 011 000	001 001 010	3
10	000 010 000	110 100 001	5
11	000 100 000	010 100 000	1
12	000 110 000	000 110 011	2
13	000 111 000	100 100 011	5
14	000 101 000	010 001 010	3
15	001 000 000	001 000 111	3
16	011 000 000	011 010 010	2
17	111 000 000	100 000 000	2
18	100 000 000	100 101 010	3
19	101 000 000	111 000 010	2
20	010 000 000	010 000 000	0
21	110 000 001	110 100 001	1
22	110 001 110	111 011 010	3
23	011 100 110	000 100 101	4
24	110 110 111	100 110 111	1
25	111 111 001	000 101 001	4
26	000 001 001	000 001 010	2
27	000 001 011	100 001 111	2
28	000 001 010	010 001 000	2
29	000 001 111	000 111 101	3
30	000 001 100	010 010 100	3
31	000 001 110	100 001 010	2
32	000 011 001	010 011 101	2
33	000 011 011	000 011 110	1
34	000 011 111	000 111 111	1
35	000 011 101	011 000 111	5
36	000 011 110	010 010 110	2
37	000 011 010	000 000 000	3
38	000 111 001	000 111 100	2
39	000 111 011	001 101 011	2
40	000 111 111	110 100 111	4
41	000 111 101	000 111 101	0
42	000 111 110	000 110 110	1
43	000 111 010	011 010 011	4
44	000 101 001	100 101 101	2
45	000 101 011	110 100 010	4
46	000 101 111	001 101 110	2
47	000 101 101	000 001 111	2
48	000 101 110	010 001 111	2
49	000 101 010	011 101 010	2
50	111 000 101	111 100 101	1
Total Kesalahan	113		

Tabel 2. Hasil simulasi kinerja kode LT

# Percobaan	Bit Informasi	Bit Hasil Dekode LT	Jumlah kesalahan dekode LT
1	000 000 001	000 000 011	1
2	000 000 011	001 001 011	2
3	000 000 111	010 000 100	3
4	000 000 100	001 001 110	4
5	000 000 101	001 010 110	4
6	000 000 010	000 000 000	1
7	000 000 110	010 001 100	3
8	000 001 000	001 001 101	3
9	000 011 000	011 100 000	5
10	000 010 000	100 011 000	3
11	000 100 000	110 100 000	2
12	000 110 000	101 000 011	6
13	000 111 000	011 110 000	3
14	000 101 000	010 001 111	5
15	001 000 000	101 010 000	2
16	011 000 000	111 000 000	1
17	111 000 000	110 001 001	3
18	100 000 000	101 000 011	3
19	101 000 000	111 001 100	3
20	010 000 000	110 000 000	1
21	110 000 001	010 000 100	2
22	110 001 110	111 111 100	4
23	011 100 110	001 001 100	4
24	110 110 111	100 010 101	3
25	111 111 001	100 001 011	5
26	000 001 001	101 011 001	3
27	000 001 011	000 001 011	2
28	000 001 010	101 001 000	3
29	000 001 111	000 001 010	2
30	000 001 100	000 001 100	0
31	000 001 110	010 101 010	3
32	000 011 001	000 000 001	2
33	000 011 011	000 101 111	3
34	000 011 111	110 110 110	5
35	000 011 101	000 001 110	3
36	000 011 110	110 110 100	5
37	000 011 010	010 011 011	2
38	000 111 001	000 110 010	3
39	000 111 011	001 011 101	4
40	000 111 111	101 001 010	6
41	000 111 101	110 100 101	4
42	000 111 110	001 010 000	5
43	000 111 010	101 111 000	3
44	000 101 001	101 000 111	6
45	000 101 011	100 100 100	4
46	000 101 111	000 001 111	2
47	000 101 101	001 101 100	2
48	000 101 110	110 001 000	3
49	000 101 010	010 000 011	4
50	111 000 101	000 000 111	3
Total Kesalahan			158

Dari hasil simulasi yang ditampilkan pada Tabel 1, tampak bahwa kode Raptor berhasil mendekodekan atau merekonstruksi bit informasi yang dikirim melalui kanal *erasure*. Namun demikian terdapat kesalahan dekode sebesar 25,1% dari keseluruhan bit yang dikirim. Begitu pula halnya dengan kode LT yang hasil simulasinya diberikan pada Tabel 2. Kode LT dapat mendekodekan dan merekonstruksi ulang simbol informasi yang dikirimkan pada kanal BEC dengan probabilitas kesalahan sebanyak 35,1%. Sejumlah simulasi menunjukkan kode LT memiliki kinerja yang sama atau lebih baik dibandingkan dengan kode Raptor, akibat perubahan matriks *G* secara acak setiap kali simulasi. Dalam simulasi yang digunakan pada penelitian ini, kanal BEC dimodelkan dengan menghapus salah satu kolom pada matriks *G* secara acak. Apabila kolom yang terhapus mengandung banyak angka "1", maka berarti *codeword* dengan distribusi *degree* yang besar berpotensi tidak dapat didekodekan atau direkonstruksi. Kode Raptor memiliki kehandalan lebih baik dari kode LT, karena *pre-code* Reed-Solomon dapat membantu mendekodekan *codeword* yang tidak bisa didekodekan dengan kode LT.

Pada gambar 23 merupakan tampilan utama dari sistem ini yang menampilkan jam, tanggal dan status dari sistem kendali dimana "I" mewakili pintu masuk air, "O" mewakili pintu keluar air dan "P" mewakili pompa air. Nilai 0 berarti tidak aktif atau sama dengan pintu tertutup dan nilai 1 berarti aktif atau sama dengan pintu terbuka. Pada gambar 26 merupakan tampilan ketika tombol back ditekan. Pada bagian ini LCD menampilkan pengaturan jadwal pengiriman paket yang digunakan dalam menit dan total merupakan waktu yang sudah dilewati dari pengiriman paket data terakhir.

IV. Kesimpulan

Hasil simulasi dengan ukuran matriks generator berukuran 15 x 30 menunjukkan bahwa kode Raptor dapat merekonstruksi ulang simbol yang dikirimkan pada kanal BEC, walaupun simbol-simbol tersebut tidak utuh tiba di penerima. Simulasi memberikan gambaran bahwa kode ini dapat mengatasi permasalahan *erasure* pada kanal. Kode Raptor menggunakan redundansi bit yang besar, misalnya dalam penelitian ini digunakan hingga 21 bit paritas untuk membentuk *codeword*. Hal ini dapat menjadi suatu kelemahan bila dipandang dari sisi efisiensi *bandwidth*, namun sebenarnya berpotensi untuk dimanfaatkan sebagai salah satu metode laju kode adaptif sesuai kondisi kanal. Dalam kondisi riil distribusi Robust Soliton mutlak diperlukan agar jumlah bit redundan dapat ditekan.

Daftar Pustaka

1. Shokrollahi A. Raptor Codes. IEEE Transactions on Information Theory. 2006; 52(6): 2551-2567.
2. Sivasubramanian B, Leib H. Fixed-Rate Raptor Codes Over Rician Fading Channels. IEEE Transaction of Vehicular Technology. 2008; 57(6): 3905-3911.
3. Etesami, O., & Shokrollahi, A. Raptor Codes on Binary Memoryless Symmetric Channels. IEEE Transactions on Information Theory. 2006; 2033-2051.
4. MacKay, D.J.C. Fountain Codes. IEE Proc.-Commun. 2005; 152:1062-1068.
5. Luby, M. LT Codes. Proc. 43rd Ann. IEEE Symposium on Foundations of Computer Science. 2002. pp.271-282
6. Moreira, J.C. and Farell, P.G. 2006. Essentials of Error-Control Coding. England : John Wiley & Sons, Ltd.
7. Proakis JG. Digital Communications. Singapore: McGraw-Hill International. 2008..